Working with DB2 UDB objects

Presented by DB2 Developer Domain

http://www7b.software.ibm.com/dmdd/

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Introduction

What this tutorial is about

This tutorial discusses data types, tables, views, and indexes as defined by DB2 Universal Database. It explains the features of these objects, how to create and manipulate them using Structured Query Language (SQL), and how they can be used in an application.

This tutorial is the fifth in a series of six tutorials that you can use to help prepare for the DB2 UDB V8.1 Family Fundamentals Certification (Exam 700). The material in this tutorial primarily covers the objectives in Section 5 of the exam, entitled "Working with DB2 UDB Objects." You can view these objectives at: *http://www.ibm.com/certify/tests/obj700.shtml*.

In this tutorial, you will learn about:

- The built-in data types provided by DB2, and which data types are appropriate to use when defining a table. (For a different treatment of data type, see the *fourth tutorial in this series*.
- The concepts of advanced data types.
- Creating tables, views, and indexes in a DB2 database.
- The features and use of unique constraints, referential integrity constraints, and table check constraints.
- How to use views to restrict access to data.
- The features use of indexes.

You do not need a copy of DB2 Universal Database to complete this tutorial. However, if you'd like, you can download a trial version of *IBM DB2 Universal Database* Enterprise Edition.

About the author

Hana Curtis is a member of the DB2 Integration team at the IBM Toronto Software Laboratory working with DB2 and WebSphere. Previously, she was a database consultant working with IBM Business Partners to enable their applications to DB2. She is an IBM Certified Solutions Expert - DB2 UDB V7.1 Database Administration for UNIX, Windows, and OS/2 and DB2 UDB V7.1 Family Application Development. Hana is one of the authors of the recently published book: *DB2 SQL Procedural Language for Linux, UNIX, and Windows* (Prentice Hall, 2003). She holds the following certifications:

- IBM Certified Solutions Expert: DB2 UDB V7.1 Database Administration for UNIX, Windows, and OS/2
- IBM Certified Solutions Expert: DB2 UDB V7.1 Family Application Development

- IBM Certified Specialist: DB2 V7.1 User

Section 2. Data types

Categories of data types

DB2 provides a rich and flexible assortment of data types. DB2 comes with basic data types such as INTEGER, CHAR, and DATE. It also includes facilities to create user-defined data types (UDTs) so that you can create complex, nontraditional data types suited to today's complex programming environments. Choosing which type to use in a given situation depends on the type and range of information that will be stored in the column.

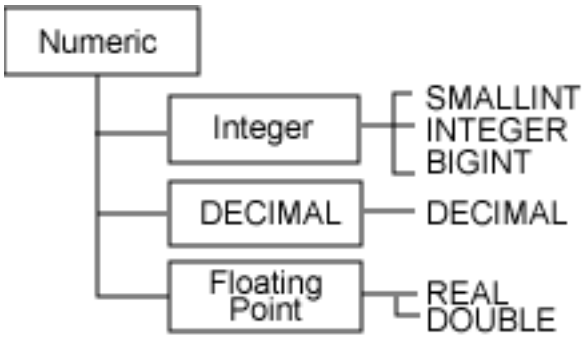
The built-in data types are categorized as follows:

- Numeric
- String
- Datetime
- Datalink

The user-defined data types are categorized as:

- User-defined distinct type
- User-defined structured type
- User-defined reference type

Numeric data types



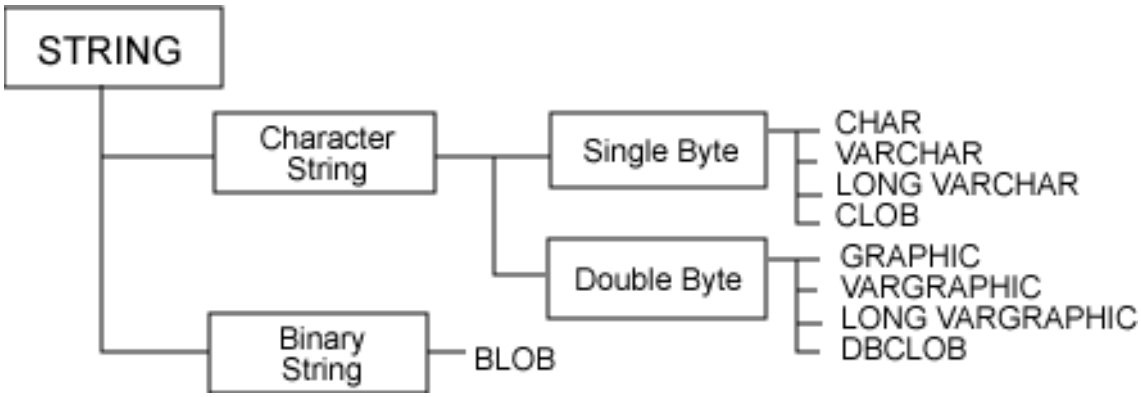
There are three categories of numeric data types, diagrammed in the figure above. These types vary in the range and precision of numeric data they can store.

- **Integer:** MALLINT, INTEGER and BIGINT are used to store integer numbers. For example, an inventory count could be defined as INTEGER. SMALLINT can store integers from -32,768 to 32,767 in 2 bytes. INTEGER can store integers from -2,147,483,648 to 2,147,483,647 in 4 bytes. BIGINT can store integers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 in 8 bytes.

- **Decimal:** DECIMAL is used to store numbers which fractional parts. To define this data type, you must specify a *precision* (p), which indicates the total number of digits, and a *scale* (s), which indicates the number of digits to the right of the decimal place. A column defined by DECIMAL(10,2) that held currency values could hold values up to 10 million dollars. The amount of storage required in the database depends on the precision and is calculated by the formula p/2 +1. So, DECIMAL(10,2) would require 10/2 + 1 or 6 bytes.

- **Floating point:** REAL and DOUBLE are used to store approximations of numbers. For example, very small or very large scientific measurements could be defined as REAL. REAL can be defined with a length between 1 and 24 digits and requires 4 bytes of storage. DOUBLE can be defined with a length of between 25 and 53 digits and requires 8 bytes of storage. FLOAT can be used as a synonym for REAL or DOUBLE.

String data types



DB2 provides several data types for storing character data or strings, diagrammed in the figure above. You will choose a data type based on the size of the string you are going to store and what data will be in the string.

The following data types are used to store single-byte character strings:

- **CHAR:** CHAR or CHARACTER is used to store fixed-length character strings up to 254 bytes. For example, a manufacturer may assign an identifier to a part with a specific length of eight characters, and will therefore store that identifier in the database as a column of CHAR(8).

- **VARCHAR:** VARCHAR is used to store variable-length character strings. For example, a manufacturer may deal with a number of a parts with identifiers of different lengths, and thus will store those identifiers as a column of VARCHAR(100). The maximum length of a VARCHAR column is 32,672 bytes. In the database, VARCHAR data only takes as much space as required.

The following data types are used to store double-byte character strings:

- **GRAPHIC:** GRAPHIC is used to store fixed-length double-byte character strings. The maximum length of a GRAPHIC column is 127 characters.

- **VARGRAPHIC:** VARGRAPHIC is used to store variable-length double-byte character strings. The maximum length of a VARGRAPHIC column is 16,336 characters.

DB2 also provides data types to store very long strings of data. All long string data types have similar characteristics. First, the data is not stored physically with the row data in the database, which means that additional processing is required to access this data. Long data types can be defined up to 2 GB in length. However, only the space required is actually used. The long data types are:

- **LONG VARCHAR**
- **CLOB** (character large object)
- **LONG VARGRAPHIC**
- **DBCLOB** (double-byte character large object)
- **BLOB** (binary large object)

Datetime data types

DB2 provides three data types to store dates and times:

- **DATE**
- **TIME**
- **TIMESTAMP**

The values of these data types are stored in the database in an internal format; however, applications can manipulate them as strings. When one of these data types is retrieved, it is represented as a character string. When updating these data types, you must enclose the value in quotation marks.

DB2 provides built-in functions to manipulate datetime values. For example, you can determine the day of the week of a date value using the `DAYOFWEEK` or `DAYNAME` functions. You can use the `DAYS` function to calculate how many days lie between two dates. DB2 also provides special registers that can be used to generate the current date, time, or timestamp based on the time-of-day clock. For example, `CURRENT DATE` returns a string representing the current date on the system.

The format of the date and time values depends on the country code of the database, which is specified when the database is created. There are several formats available: ISO, USA, EUR, and JIS. For example, if your database is using the USA format, the format of date values would be mm/dd/yyyy. You can change the format by using the

DATETIME option of the BIND command when creating your application.

There is a single format for the TIMESTAMP data type. The string representation is yyyy-mm-dd-hh.mm.ss.nnnnnn.

Datalinks

DB2 provides the DATALINK data type to manage external files. A DATALINK column allows you to store a reference to a file outside the database. These files can reside in a file system on the same server as the database or on a remote server. DB2 provides facilities that allow applications to access these files securely.

To insert values into a DATALINK column, you must use the built-in function DLVALUE. DLVALUE requires several parameters, which tell DB2 the name and location of the file to which you're linking. To retrieve data from the DATALINK column, DB2 provides several functions; the one you'll use will depend on the information you require.

User-defined data types

DB2 allows you to define data types that suit your application. There are three user-defined data types:

- **User-defined distinct types:** You can define a new data type based on a built-in type. This new type will have the same features of the built-in type, but you can use it to ensure that only values of the same type are compared. For example, you can define a Canadian dollar type (CANDOL) and a US dollar type (USADOL) both based on DECIMAL(10,2). Both types are based on the same built-in type, but you won't be able to compare them unless a conversion function is applied. The following CREATE TYPE statements will create the CANDOL and USADOL UDTs:

  ```
  CREATE DISTINCT TYPE CANDOL AS DECIMAL(10,2) WITH COMPARISONS
  CREATE DISTINCT TYPE USADOL AS DECIMAL(10,2) WITH COMPARISONS
  ```

  DB2 automatically generates functions to perform casting between the base type and the distinct type, and comparison operators for comparing instances of the distinct type. The following statements show how to create a table with a column of CANDOL type and then insert data into the table using the CANDOL casting function.

  ```
  CREATE TABLE ITEMs (ITEMID CHAR(5), PRICE CANDOL )
  INSERT INTO ITEMs VALUES('ABC11',CANDOL(30.50) )
  ```

- **User-defined structured types:** You can create a type that consists of several

columns of built-in types. You can then use this structured type when creating a table. For example, you can create a structured type named ADDRESS that contains data for street number, street name, city, etc. Then you can use this type when defining other tables, such as employees or suppliers, since the same data is required for both. Also, structured types can have subtypes in a hierarchical structure. This allows objects that belong to a hierarchy to be stored in the database.

- **User-defined reference types:** When using structured types, you can define references to rows in another table using reference types. These references appear similar to referential constraints; however, they do not enforce relationships between the tables. References in tables allow you to specify queries in a different way.

User-defined structured and reference types are an advanced topic; the information presented here serves only as an introduction to these types.

DB2 Extenders

DB2 Extenders provide support for complex, nontraditional data types. They are packaged separately from the DB2 server code and must be installed on the server and into each database that will use the data type.

There are many DB2 Extenders available from IBM and from independent software vendors. The first four extenders provided by IBM were for storing audio, video, image, and text data. For example, the DB2 Image Extender can be used to store an image of a book cover and the DB2 Text Extender can be used to store the text of a book. Now there are several other extenders available, including the XML Extender, which allows you to manage XML documents in a DB2 database.

DB2 Extenders are implemented using the features of user-defined types and user-defined functions (UDFs). Each extender comes with one or more UDT, UDFs for operating on the UDT, specific application programming interfaces (APIs), and perhaps other tools. For example, the DB2 Image Extender includes:

- The DB2IMAGE UDT
- UDFs to insert/retrieve from a DB2IMAGE column
- APIs to search based on characteristics of images

Before using these data types, you must install the extender support into the database. The installation process for each extender defines the required UDTs and UDFs in the database. Once you've done this, you can use the UDTs when defining a table and the UDFs when working with data. (For more on DB2 Extenders, see the *first tutorial in this series*.

Section 3. Tables

What are tables?

All data is stored in tables in the database. A table consists of one or more columns of various data types. The data is stored in rows or records.

Tables are defined using the CREATE TABLE SQL statement. DB2 also provides a GUI tool for creating tables, which will create a table based on information you specify. It will also generate the CREATE TABLE SQL statement that can be used in a script or application program at a later time.

A database has a set of tables, called the *system catalog tables,* which hold information about all the objects in the database. The catalog table SYSCAT.TABLES contains a row for each table defined in the database. SYSCAT.COLUMNS contains a row for each column of each table in the database. You can look at the catalog tables using SELECT statements, just like any other table in the database; however, you cannot modify the data using INSERT, UPDATE, or DELETE statements. The tables are automatically updated as a result of data definition statements (DDL), such as CREATE, and other operations, such as RUNSTATS.

Creating a table

The CREATE TABLE SQL statement is used to define a table in the database. The following statement will create a simple table named BOOKS that contains three columns:

```
CREATE TABLE BOOKS ( BOOKID INTEGER,
                     BOOKNAME VARCHAR(100),
                     ISBN CHAR(10) )
```

You can also use the CREATE TABLE SQL statement to create a table that is like another table or view in the database:

```
CREATE TABLE MYBOOKS LIKE BOOKS
```

This statement creates a table with the same columns as the original table or view. The columns of the new table have the same names, data types, and nullability attributes as the columns in the old one. You can also specify clauses that will copy other attributes, like column defaults and identify attributes.

There are many options available for the CREATE TABLE statement (they'll be presented in the following sections as new concepts are introduced). The details of the CREATE TABLE SQL statement can be found in the SQL Reference (see Resources on page 25).

Once you've created a table, there are several ways to populate it with data. The INSERT statement allows you to insert a row or several rows of data into the table. DB2 also provides utilities to insert large amounts of data from a file. The IMPORT utility inserts rows using INSERT statements. It is designed for loading small amounts of data into the database. The LOAD utility inserts rows directly onto data pages in the database and is therefore much faster than the IMPORT utility. It is intended for loading large volumes of data.

Where is a table stored in the database?

Tables are stored in the database in *tablespaces.* Tablespaces have physical space allocated to them. You must create the tablespace before creating the table.

When you create a table, you can let DB2 place the table in a default tablespace, or you can specify the tablespace in which you'd like the table to reside. The following CREATE TABLE statement places the BOOKS table in the BOOKINFO tablespace.

```
CREATE TABLE BOOKS ( BOOKID INTEGER,
                     BOOKNAME VARCHAR(100),
                     ISBN CHAR(10) )
              IN BOOKINFO
```

Although we will not discuss tablespaces here in detail, it is important to understand that defining tablespaces appropriately will have an effect on the performance and maintainability of the database. For more information on tablespaces, check out the *second tutorial in this series*.

Altering a table

You can change certain characteristics of a table using the ALTER TABLE SQL statement. For instance, you can:

- Add one or more columns
- Add or drop a primary key
- Add or drop one or more unique or referential constraints
- Add or drop one or more check constraints
- Change the length of a VARCHAR column

For example, the following statement adds a column called BOOKTYPE to the BOOKS table:

```
ALTER TABLE BOOKS ADD BOOKTYPE CHAR(1)
```

Certain characteristics of a table cannot be changed. For example, you cannot remove

a column from a table. Also, you cannot change the tablespace in which the table resides. To change characteristics such as these, you must save the table data, drop the table, and recreate it.

Dropping a table

The DROP TABLE statement removes a table from the database, deleting the data and the table definition. If there are indexes or constraints defined on the table, they are dropped as well.

The following DROP TABLE statement deletes the BOOKS table from the database:

```
DROP TABLE BOOKS
```

NOT NULL, DEFAULT, and GENERATED column options

The columns of a table are specified in the CREATE TABLE statement by a column name and data type. The columns can have additional clauses specified that restrict the data in the column.

By default, a column allows null values. If you do not want to allow null values, you can specify the NOT NULL clause for the column. You can also specify a default value using the WITH DEFAULT clause and a default value. The following CREATE TABLE statement creates a table BOOKS where the BOOKID column does not allow null values and the default value for BOOKNAME is TBD.

```
CREATE TABLE BOOKS ( BOOKID INTEGER NOT NULL,
                     BOOKNAME VARCHAR(100) WITH DEFAULT 'TBD',
                     ISBN CHAR(10) )
```

In the BOOKS table, the BOOKID is a unique number assigned to each book. Rather than have the application generate the identifier, we can specify that DB2 is to generate a BOOKID using the GENERATED ALWAYS AS IDENTITY clause:

```
CREATE TABLE BOOKS ( BOOKID INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY
                                             (START WITH 1, INCREMENT BY 1),
                     BOOKNAME VARCHAR(100) WITH DEFAULT 'TBD',
                     ISBN CHAR(10) )
```

GENERATED ALWAYS AS IDENTITY causes a BOOKID to be generated for each record. The first value generated will be 1 and succeeding values will be generated by incrementing the previous value by 1.

You can also use the GENERATED ALWAYS option to have DB2 calculate the value of a column automatically. The following example defines a table called AUTHORS, with counts for fiction and nonfiction books. The TOTALBOOKS column will be calculated by adding the two counts.

```
CREATE TABLE AUTHORS (AUTHORID INTEGER NOT NULL PRIMARY KEY,
                      LNAME VARCHAR(100),
                      FNAME VARCHAR(100),
                      FICTIONBOOKS INTEGER,
                      NONFICTIONBOOKS INTEGER,
                      TOTALBOOKS INTEGER GENERATED ALWAYS
                                 AS (FICTIONBOOKS + NONFICTIONBOOKS) )
```

Section 4. Constraints

What are constraints?

DB2 provides several ways to control what data can be stored in a column. These features are called *constraints* or *rules* that the database manager enforces on a data column or set of columns.

DB2 provides three types of constraints:

- *Unique* constraints, which are used to ensure that values in a column are unique.
- *Referential integrity* constraints, which are used to define relationships between tables and ensure that these relationships remain valid.
- *Table check* constraints, which are used verify that column data does not violate rules defined for the column.

We'll discuss each type of constraint in detail in subsequent panels.

Unique constraints

Unique constraints are used to ensure that values in a column are unique. Unique constraints can be defined over one or more columns. Each column included in the unique constraint must be defined as `NOT NULL`.

Unique constraints can be defined either as the `PRIMARY KEY` or `UNIQUE` constraint. These can be defined when a table is created as part of the `CREATE TABLE` SQL statement or added after the table is created using the `ALTER TABLE` statement.

When do you define a `PRIMARY KEY`, and when do you define a `UNIQUE` key? This depends on the nature of the data. In the previous example, the BOOKS table has a BOOKID which is used to uniquely identify a book. This value is also used in other tables that contain information related to this book. In this case, you would define BOOKID as a primary key. DB2 allows only one primary key to be defined on a table.

The ISBN number column needs to be unique but is not a value that is otherwise referenced in the database. In this case, the ISBN column can be defined as `UNIQUE`.

```
CREATE TABLE BOOKS (BOOKID INTEGER NOT NULL PRIMARY KEY,
                    BOOKNAME VARCHAR(100),
                    ISBN CHAR(10) NOT NULL CONSTRAINT BOOKSISBN UNIQUE )
```

The `CONSTRAINT` keywords allows you to specify a name for the constraint. In this example, the name of the unique constraint is BOOKSISBN. Use this name in the `ALTER TABLE` statement if you want to drop the specific constraint.

DB2 allows only one primary key to be defined on a table; however, you can define multiple unique constraints.

Whenever you define a PRIMARY or UNIQUE key on a column, DB2 creates a unique index to enforce uniqueness on the column. DB2 will not allow you to create duplicate unique constraints or duplicate indexes. For example, the following statement against the BOOKS table will fail:

```
ALTER TABLE BOOKS ADD  CONSTRAINT UNIQUE (BOOKID)
```

Referential integrity constraints

Referential integrity constraints are used to define relationships between tables. Suppose we have one table that holds information about authors, and another table that lists the books that those authors have written. There is a relationship between the BOOKS table and the AUTHORS table -- each book has an author and that author must exist in the AUTHOR table. Each author had a unique identifier stored in the AUTHORID column. The AUTHORID is used in the BOOKS table to identify the author of each book. To define this relationship, define the AUTHORID column of the AUTHORS table as a primary key and then define a foreign key on the BOOKS table to establish the relationship with the AUTHORID column in the AUTHORS table, like so:

```
CREATE TABLE AUTHORS (AUTHORID INTEGER NOT NULL PRIMARY KEY,
                      LNAME VARCHAR(100),
                      FNAME VARCHAR(100))
CREATE TABLE BOOKS (BOOKID INTEGER NOT NULL PRIMARY KEY,
                    BOOKNAME VARCHAR(100),
                    ISBN CHAR(10),
                    AUTHORID INTEGER REFERENCES AUTHORS)
```

The table that has a primary key that relates to another table -- AUTHOR, here -- is called a *parent table.* The table to which the parent table relates -- BOOKS, here -- is called a *dependent table.* You may define more than one dependent table on a single parent table.

You can also define relationships between rows of the same table. In such a case, the parent table and dependent tables are the same table.

When you define referential constraints on a set of tables, DB2 enforces referential integrity rules on those tables when update operations are performed against them:

- DB2 ensures that only valid data is inserted into columns where referential integrity constraints are defined. This means that you must always have a row in the parent table with a key value that is equal to the foreign key value in the row that you are inserting into a dependent table. For example, if a new book is being inserted into the BOOKS table with an AUTHORID of 437, then there must already be a row in the AUTHORS table where AUTHORID is 437.

- DB2 also enforces rules when rows that have dependent rows in a dependent table

are deleted from a parent table. The action DB2 takes depends on the delete rule defined on the table. There are four rules that can be specified: RESTRICT, NO ACTION, CASCADE and SET NULL.

- If RESTRICT or NO ACTION is specified, DB2 does not allow the parent row to be deleted. The rows in dependent tables must be deleted before the row in the parent table. This is the default, so this rule applies to the AUTHORS and BOOKS tables as we've defined them so far.
- If CASCADE is specified, then deleting a row from the parent table automatically also deletes dependent rows in all dependent tables.
- If SET NULL is specified, then the parent row is deleted from the parent table and the foreign key value in the dependent rows is set to null (if nullable).

- When updating key values in the parent table, there are two rules that can be specified: RESTRICT and NO ACTION. RESTRICT will not allow a key value to be updated if there are dependent rows in a dependent table. NO ACTION causes the update operation on a parent key value to be rejected if, at the end of the update, there are dependent rows in a dependent table that do not have a parent key in the parent table.

Table check constraints

Table check constraints are used to restrict the values in a certain column of a table. DB2 will ensure that the constraint is not violated during inserts and updates.

Suppose that we add a column to the BOOKS table for a book type, and the values that we wish to allow are F (fiction) and N (nonfiction). We can add a column BOOKTYPE with a check constraint as follows:

```
ALTER TABLE BOOKS ADD BOOKTYPE CHAR(1) CHECK (BOOKTYPE IN ('F','N') )
```

You can define check constraints when you create the table or add them later using the `ALTER TABLE` SQL statement. You can modify check constraints by dropping and then recreating them using the `ALTER TABLE` SQL statement.

Section 5. Views

What are views?

Views allow different users or applications to look at the same data in different ways. This not only makes the data simpler to access, but it can also be used to restrict which rows and columns can be viewed or updated by certain users.

For example, suppose that a company has a table containing information about its employees. A manager needs to see address, telephone number, and salary information about his employees only, while a directory application needs to see all employees in the company along with their address and telephone numbers, but not their salaries. You can create one view that shows all the information for the employees in a specific department, and another that shows only the name, address, and telephone number of all employees.

To the user, a view just looks like a table. Except for the view definition, a view does not take up space in the database; the data presented in a view is derived from another table. You can create a view on an existing table (or tables), on another view, or some combination of the two. A view defined on another view is called a *nested view.*

You can define a view with column names that are different than the corresponding column names of the base table. You can also define views that check to see if data inserted or updated stays within the conditions of the view.

The list of views defined in the database is stored in the system catalog table SYSIBM.SYSVIEWS, which also has a view defined on it called SYSCAT.VIEWS. The system catalog also has a SYSCAT.VIEWDEP which, for each view defined in the database, has a row for each view or table dependent on that view. Also, each view has an entry in SYSIBM.SYSTABLES and entries in SYSIBM.SYSCOLUMNS, since views can be used just like tables.

Creating a view

The `CREATE VIEW` SQL statement is used to define a view. A `SELECT` statement is used to specify which rows and columns will be presented in the view.

For example, imagine that we want to create a view that will show only the nonfiction books in our BOOKS table:

```
CREATE VIEW NONFICTIONBOOKS AS
       SELECT * FROM BOOKS WHERE BOOKTYPE = 'N'
```

Note that after we define this view, there will be entries for it in SYSCAT.VIEWS, SYSCAT.VIEWDEP, and SYSCAT.TABLES.

To define column names in the view that are different from those in the base table, you can specify them in the CREATE VIEW statement. The following statement creates a MYBOOKVIEW view that contains two columns: TITLE, which represents the BOOKNAME column, and TYPE, which represents the BOOKTYPE column.

```
CREATE VIEW MYBOOKVIEW (TITLE,TYPE) AS
       SELECT BOOKNAME,BOOKTYPE FROM BOOKS
```

The DROP VIEW SQL statement is used to drop a view from the database. If you drop a table or another view on which a view is based, the view remains defined in the database but becomes inoperative. The VALID column of SYSCAT.VIEWS indicates whether a view is valid (Y) or not (X). Even if you recreate the base table, the orphaned view will remain invalid; you will have to recreate it as well.

You can drop the NONFICTIONBOOKS view from the database like so:

```
DROP VIEW NONFICTIONBOOKS
```

You cannot modify a view; to change a view definition, you must drop it and recreate it. You would use the ALTER VIEW statement provided only to modify reference types, and we will not discuss it here.

Read-only and updatable views

When you create a view, you can define it as either a *read-only* view or as an *updatable* view. The SELECT statement of a view determines whether the view is read-only or updatable. Generally, if the rows of a view can be mapped to rows of the base table, then the view is updatable. For example, the view NONFICTIONBOOKS as we defined it in the previous example is updatable, because each row in the view is a row in the base table.

The rules for creating updatable views are complex and depend on the definition of the query. For example, views that use VALUES, DISTINCT, or JOIN features are not updatable. You can easily determine whether a view is updatable by looking at the READONLY column of SYSCAT.VIEWS: Y means it is ready-only and N means it is not.

The detailed rules for creating updatable views are documented in the DB2 SQL Reference (see Resources on page 25).

Views with check option

The NONFICTIONBOOKS view defined previously includes only the rows where the BOOKTYPE is N. If you insert a row where the BOOKTYPE is F into the view, DB2 will

insert the row into the base table BOOKS. However, if you then select from the view, the newly inserted row cannot be seen through the view. If you do not want to allow a user to insert rows that are outside the scope of the view, you can define the view with the *check option.* Defining a view using `WITH CHECK OPTION` tells DB2 to check that statements using the view satisfy the conditions of the view.

The following statement defines a view using `WITH CHECK OPTION`:

```
CREATE VIEW NONFICTIONBOOKS AS
       SELECT * FROM BOOKS WHERE BOOKTYPE = 'N'
       WITH CHECK OPTION
```

This view still restricts the user to seeing only non-fiction books; in addition, it also prevents the user from inserting rows that do not have a value of N in the BOOKTYPE column, and updating the value of the BOOKTYPE column in existing rows to a value other than N. The following statements, for instance, will no longer be allowed:

```
INSERT INTO NONFICTIONBOOKS VALUES (...,'F');
UPDATE NONFICTIONBOOKS SET BOOKTYPE = 'F' WHERE BOOKID = 111
```

Nested views with check option

When defining nested views, the check option can be used to restrict operations. However, there are other clauses you can specify to define how the restrictions are inherited. The check option can be defined either as `CASCADED` or `LOCAL`. `CASCADED` is the default if the keyword is not specified. To explain the differences between the behavior of `CASCADED` and `LOCAL`, we need to look at several possible scenarios.

When a view is created `WITH CASCADED CHECK OPTION`, all statements executed against the view must satisfy the conditions of the view and all underlying views -- even if those views were not defined with the check option. Suppose that the view NONFICTIONBOOKS is created without the check option, and we also create a view NONFICTIONBOOKS1 based on the view NONFICTIONBOOKS using the CASCADED keyword:

```
CREATE VIEW NONFICTIONBOOKS AS
       SELECT * FROM BOOKS WHERE BOOKTYPE = 'N'
CREATE VIEW NONFICTIONBOOKS1 AS
       SELECT * FROM NONFICTIONBOOKS WHERE BOOKID > 100
       WITH CASCADED CHECK OPTION
```

The following `INSERT` statements would not be allowed because they do not satisfy the conditions of at least one of the views:

```
INSERT INTO NONFICTIONBOOKS1 VALUES( 10,..,'N')
INSERT INTO NONFICTIONBOOKS1 VALUES(120,..,'F')
INSERT INTO NONFICTIONBOOKS1 VALUES( 10,..,'F')
```

However, the following `INSERT` statement *would* be allowed because it satisfies the

conditions of both of the views:

```
INSERT INTO NONFICTIONBOOKS1 VALUES(120,...,'N')
```

Next, suppose we create a view NONFICTIONBOOKS2 based on the view NONFICTIONBOOKS using `WITH LOCAL CHECK OPTION`. Now, statements executed against the view need only satisfy conditions of views that have the check option specified.

```
CREATE VIEW NONFICTIONBOOKS AS
       SELECT * FROM BOOKS WHERE BOOKTYPE = 'N'
CREATE VIEW NONFICTIONBOOKS2 AS
       SELECT * FROM NONFICTIONBOOKS WHERE BOOKID > 100
       WITH LOCAL CHECK OPTION
```

In this case, the following `INSERT` statements would not be allowed because they do not satisfy the `BOOKID > 100` condition of the NONFICTIONBOOKS2 view.

```
INSERT INTO NONFICTIONBOOKS2 VALUES(10,..,'N')
INSERT INTO NONFICTIONBOOKS2 VALUES(10,..,'F')
```

However, the following `INSERT` statements *would* be allowed even though the value N does not satisfy the `BOOKTYPE = 'N'` condition of the NONFICTIONBOOKS view.

```
INSERT INTO NONFICTIONBOOKS2 VALUES(120,..,'N')
INSERT INTO NONFICTIONBOOKS2 VALUES(120,..,'F')
```

Section 6. Indexes

What are indexes?

An index is an ordered list of the key values of a column or columns of a table. There are two reasons why you might create an index:

- To ensure uniqueness of values in a column or columns.
- To improve performance of queries against the table. The DB2 optimizer will use indexes to improve performance when performing queries, or to present results of a query in the order of the index.

Indexes can be defined as *unique* or *nonunique.* Nonunique indexes allow duplicate key values; unique indexes allow only one occurrence of a key value in the list. Unique indexes do allow a single null value to be present. However, a second null value would cause a duplicate and therefore is not allowed.

Indexes are created using the `CREATE INDEX` SQL statement. Indexes are also created implicitly in support of a `PRIMARY KEY` or `UNIQUE` constraint. When a unique index is created, the key data is checked for uniqueness and the operation will fail if duplicates are found.

Indexes are created as ascending, descending, or bidirectional. The option you choose depends on how the application will access the data.

Creating indexes

In our example, we have a primary key on the BOOKID column. Often, users will conduct searches on the book title, so an index on BOOKNAME would be appropriate. The following statement creates a nonunique ascending index on the BOOKNAME column:

```
CREATE INDEX IBOOKNAME ON BOOKS (BOOKNAME)
```

The index name, IBOOKNAME, is used to create and drop the index. Other than that, the name is not used in queries or updates to the table.

By default, an index is created in ascending order, but you can also create indexes that are descending. You can even specify different orders for the columns in the index. The following statement defines an index on the AUTHORID and BOOKNAME columns. The values of the AUTHORID column are sorted in descending order, and the values of the BOOKNAME column are sorted in ascending order within the same AUTHORID.

```
CREATE INDEX I2BOOKNAME ON BOOKS (AUTHOID DESC, BOOKNAME ASC)
```

When an index is created in a database, the keys are stored in the specified order. The index helps improve performance of queries requiring the data in the specified order. An ascending index is also used to determine the result of the MIN column function; a descending index is used to determine the result of the MAX column function. If the application needs the data to be ordered in the opposite sequence to the index as well, DB2 allows the creation of a bidirectional index. A bidirectional index eliminates the need to create an index in the reverse order, and it eliminates the need for the optimizer to sort the data in the reverse order. It also allows the efficient retrieval of MIN and MAX functions values. To create a bidirectional index, specify the ALLOW REVERSE SCANS option on the CREATE INDEX statement, like so:

```
CREATE INDEX BIBOOKNAME ON BOOKS (BOOKNAME) ALLOW REVERSE SCANS
```

DB2 will not allow you to create multiple indexes with the same definition. This applies even to indexes that you create implicitly in support of a primary key or unique constraint. So, since the BOOKS table already has a primary key defined on the BOOKID column, attempting to create an index on BOOKID column will fail.

Creating an index can take a long time. DB2 must read each row to extract the keys, sort those keys, and then write the list to the database. If the table is large, then a temporary tablespace is used sort the keys.

The index is stored in a tablespace. If your table resides in a database-managed tablespace, you have the option of separating the indexes into a separate tablespace. This must be defined when you create the table, using the INDEXES IN clause. The location of a table's indexes is set when the table is created and cannot be changed unless the table is dropped and recreated.

Of course, DB2 also provides the DROP INDEX SQL statement to remove an index from the database. There is no way to modify an index. If you need to change an index -- to add another column to the key, for example -- you will have to drop and re-create it.

Clustering indexes

You can create one index on each table as the *clustering index.* A clustering index is useful when the table data is often referenced in a particular order. The clustering index defines the order in which data is stored in the database. During inserts, DB2 attempts to place new rows close to rows with similar keys. Then, during queries requiring data in the clustering index sequence, the data can be retrieved faster.

To create an index as the clustering index, specify the CLUSTER clause on the CREATE INDEX statement.

```
CREATE INDEX IAUTHBKNAME ON BOOKS (AUTHORID,BOOKNAME) CLUSTER
```

This statement creates an index on the AUTHORID and BOOKNAME columns as the

clustering index. This index would improve the performance of queries written to list authors and all the books that they have written.

Using included columns in indexes

When creating an index, you have the option to include extra column data that will be stored with the key but that will not actually be part of the key itself and will not be sorted. The main reason for including additional columns in an index is to improve performance of certain queries: with this data already available in the index page, DB2 will not need to access the data page to fetch it. Included columns can only be defined for unique indexes. However, the included columns are not considered when enforcing uniqueness of the index.

Suppose that we often need to get a list of book names ordered by BOOKID. The query would look like this:

```
SELECT BOOKID,BOOKNAME FROM BOOK ORDER BY BOOKID
```

We could create an index that could possibly improve performance, like so:

```
CREATE UNIQUE INDEX IBOOKID ON BOOKS (BOOKID) INCLUDE(BOOKNAME)
```

As a result, all the data required for the query result is present in the index and no data pages need to be retrieved.

So why not just include all the data in the indexes? First of all, this would require more physical space in the database, because the table data would essentially be duplicated in the index. Second, all the copies of the data would need to be updated whenever the data value is updated, and this would be a significant overhead in a database where many updates occur.

What indexes should I create?

Here are some considerations when creating indexes.

- Since indexes are a permanent list of the key values, they require space in the database. So creating many indexes will require more storage space in your database. The amount of space required is determined by the length of the key columns. DB2 provides a tool to help you estimate the size of an index.

- Indexes are additional copies of the values, so they must be updated if the data in the table is updated. If table data is frequently updated, consider what impact additional indexes will have on update performance.

- Indexes will significantly improve performance of queries when defined on the appropriate columns.

DB2 provides a tool called the Index Advisor to help you determine which indexes to define. The Index Advisor allows you to specify the workload that will be executed against a table, and it will recommend indexes to be created on the table.

Section 7. Summary

Summary

This tutorial discussed the features of data types, tables, views, and indexes defined in a DB2 Universal Database. It also showed you how to use the CREATE, ALTER, and DROP statements to manage these objects.

DB2 provides a rich and flexible set of data types. Data types are grouped into built-in data types and user-defined data types. Built-in data types provided by DB2 are:

- Numeric: INTEGER,BIGINT,SMALLINT,DECIMAL,REAL,DOUBLE and FLOAT
- String: CHAR, VARCHAR, LONG VARCHAR, CLOB, GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, DBLOB, BLOB
- Datetime: DATE, TIME, TIMESTAMP
- Datalinks: DATALINK

DB2 also provides facilities to create advanced data types:

- User-defined distinct type
- User-defined structured type
- User-defined reference type

DB2 Extenders are an application of user-defined types. There are various DB2 Extenders available from IBM and other software vendors. Some extenders available are text, audio, video, image and XML.

Tables hold the data in the database. The columns of a table are defined by data types. Constraints can be defined on the table to provide data validation. DB2 provides three types of constraints:

- Unique constraints are used to ensure that values in a column are unique.
- Referential integrity constraints are used to define relationships between tables and ensure that these relationships remain valid.
- Table check constraints are used verify that column data does not violate rules defined for the column.

Views allow different users or applications to look at the same data in different ways. This not only makes the data simpler to access, but also can be used to restrict which rows and columns can be viewed or updated by certain users. Defining a view using WITH CHECK OPTION tells DB2 to check that updates against the view satisfy the conditions of the view. This data validation can be enforced even when nested views are specified.

An index is an ordered list of the key values of a column or columns of a table. Indexes are created to ensure uniqueness of values in a column or columns and/or improve performance of queries against the table. The DB2 optimizer chooses to use indexes in

performing queries to find the required rows faster. DB2 provides the Index Advisor to help determine which indexes to create for a specified workload.

Resources

The CREATE, ALTER, DROP and all other SQL statements are documented in:

- *IBM DB2 Universal Database SQL Reference Volume 2, Version 8*, SC09-4845-00. International Business Machines Corporation, 2002.

For more information on the DB2 Family Fundamentals Exam 700:

- *IBM Data Management Skills* information.
- Download a *self-study course for experienced database administrators (DBAs)* to quickly and easily gain skills in DB2 UDB.
- Download a *self study course for experienced relational database programmers* who would like to know more about DB2.
- *General Certification Information*, including some book suggestions, exam objectives, courses

Check out the other parts of the DB2 V8.1 Family Fundamentals Certification Prep series:

- *DB2 V8.1 Family Fundamentals Certification Prep, Part 1 of 6: DB2 Planning*
- *DB2 V8.1 Family Fundamentals Certification Prep, Part 2 of 6: DB2 Security*
- *DB2 V8.1 Family Fundamentals Certification Prep, Part 3 of 6: Accessing DB2 UDB Data*
- *DB2 V8.1 Family Fundamentals Certification Prep, Part 4 of 6: Working with DB2 UDB Data*
- *DB2 V8.1 Family Fundamentals Certification Prep, Part 6 of 6: Data Concurrency*

Feedback

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT style sheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.